

A Semantic-Based Approach for Detecting and Decomposing God Classes

Jun H. Lee
Sogang University
Seoul, S. Korea
zuna@sogang.ac.kr

Donghun Lee
Sogang University
Seoul, S. Korea
freedust@sogang.ac.kr

Dae-Kyoo Kim
Oakland University
MI, USA
kim2@oakland.edu

Sooyong Park
Sogang University
Seoul, S. Korea
sypark@sogang.ac.kr

ABSTRACT

Cohesion is a core design quality that has a great impact on posterior development and maintenance. By the nature of software, the cohesion of a system is diminished as the system evolves. God classes are code defects resulting from software evolution, having heterogeneous responsibilities highly coupled with other classes and often large in size, which makes it difficult to maintain the system. The existing work on identifying and decomposing God classes heavily relies on internal class information to identify God classes and responsibilities. However, in object-oriented systems, responsibilities should be analyzed with respect to not only internal class information, but also method interactions. In this paper, we present a novel approach for detecting God classes and decomposing their responsibilities based on the semantics of methods and method interactions. We evaluate the approach using JMeter v2.5.1 and the results are promising.

Keywords

Bad smell, God class, Large class, reengineering, refactoring, semantic analysis.

1. INTRODUCTION

Object-oriented development (OOD) is responsibility-driven. A class is assigned a single responsibility to carry out its intended purpose, having high cohesion. However, by the nature of software evolution, a single responsibility is often diminished with other responsibilities mixed by changes, which decreases the cohesion of the class and further that of the system as a whole. Such a class is desired to be restructured.

Software evolution is often ad-hoc, which makes it difficult to identify classes needing refactoring. Class size is often

used as an initial screening [1]. However, even though class size is small, the class might still need refactoring if it involves multiple responsibilities. High fan-in and fan-out is another symptom of God classes [2]. However, that is not always the case, for example facade or proxy classes whose the main responsibility is delegation. Another symptom is high complexity of methods [3]. A sorting class often involves complex methods, but is generally not large. As such, identifying refactoring needs can be subjective depending on the type of the system and developer's experience and requires techniques that enable systematic detection in consideration of various aspects.

God classes (also known as Large classes or Blobs) are code defects resulting from software evolution, having diverse responsibilities highly coupled with other classes and often large in size, which makes it difficult to maintain the system. Thus, the more God classes exist, the lower cohesion is. God classes might be inherent from design during development, which is a design defect [4].

There is some work on decomposing God classes (also known as class extraction or refactoring). The general approach of the existing work is using internal class information such as attribute-method relationships and internal method calls to identify class responsibilities [5, 2, 6, 7, 8]. However, in object-oriented systems, identifying responsibilities solely based on internal class information is very limited without considering interaction behaviors. More recent work makes use of semantic similarity of methods captured in in-line comments and identifier names, assuming that the necessary information is sufficiently available [9, 10, 11].

A key in decomposing responsibilities is to derive precise semantics of methods, so that homogeneous methods can be identified and grouped together into a separate class. In this paper, we present a semantic-based approach for detecting God classes and identifying their responsibilities based on semantic similarity of methods. Semantic similarity of methods is measured based on 1) inter-class interactions of methods, 2) intra-class interactions of methods, and 3) types of class relationships. We adopt the taxonomy by Resnik [12] for analyzing inter-class interactions of methods. The results of the taxonomy are refined by considering intra-class interactions of methods and further refined using types of class

relationships. The refined results are used for detecting God classes using weighted graphs and decomposing their responsibilities. We evaluate the presented approach using JMeter v2.5.1, a widely used open source application for load testing and measuring server performance and the results are promising. We design the approach to support its use at both design level and code level.

The remainder of the paper is organized as follows. Section 2 discusses an overview of related work. Section 3 gives an overview of the presented approach. Section 4 describes a structural taxonomy for measuring and refining semantic similarity of methods. Section 5 presents detecting God classes and identifying and decomposing their responsibilities. Section 6 evaluates the presented approach using JMeter v2.5.1 and the paper is concluded in Section 7.

2. RELATED WORK

There is much work on cohesion metrics. Chidamber and Kemerer presented Lack of Cohesion Metric (LCOM1) for measuring the number of the method pairs that reference no common attributes [13]. The higher the number of methods pairs, the lower cohesion. Revising LCOM1, they presented LCOM2 to measure class cohesion by subtracting the number of the method pairs that share attributes from LCOM1 [3]. Li and Henry [14] redefine the concept of LCOM by defining sets of methods that share an attribute. A method that shares an attribute with any method in a set becomes a member of the set. To this end, the resulting sets are completely disjoint and the number of the resulting sets indicates the cohesiveness of the class. That is, the higher the number of sets, the lower cohesiveness. Hitz and Montazeri [15] represent LCOM by Li and Henry using undirected graphs where a node represents a method and an edge represents attribute sharing by the paired methods. Cohesiveness is then measured by the number of resulting graphs, which is known as LCOM3. They also propose LCOM4 to take into account indirect reference to attributes. An edge is established between a method having a direct reference to an attribute and a method invoking the directly referencing method. Hendersen-Sellers [16] proposes LCOM5 which measures cohesion based on the number of referenced attributes. The higher cohesion, the larger the number of referenced variables. Bieman and Kang [17] proposed Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC) based on direct and indirect connectivity among methods. Marcus *et al.* [18] present Conceptual Cohesion of Classes (C3) to measure method similarity based on textual coherence of in-line comments and identifier names in source code. Briand *et al.* [19] present Ratio of Cohesive Interaction (RCI) which measures cohesiveness of a class as a ratio of the number of current interactions between method-to-data and data-to-data over the total number of possible interactions.

Several researchers use a property-based approach for decomposing God classes (e.g., see [5, 8, 6, 7]). The general approach is that the methods of a God class are measured for their similarity based on method-attribute relationships (e.g., methods sharing an attribute) and method-method relationships (e.g., method calls) using metrics (e.g., jaccard similarity metric [20]). The resulting similarity is then used as a basis for decomposing the God class. Specifically, Si-

mon *et al.* [5] measure similarity of methods based on the distance of attribute use and method call. Distance is short if one is dedicated to another (e.g., an attribute is used only in one method). Extending the work by Simon *et al.*, Fokaefs and Tsantalis [8] use a clustering algorithm for decomposing properties of a God class. Cassell *et al.* [7] use call graphs for presenting the relationship of methods and attributes and employee the Girvan-Newman betweenness clustering algorithm [21] for decomposing a Large class. Extending the property-based approach, Bavota *et al.* [11] make use of identifier names and in-line comments to measure similarity of methods using the LSI algorithm [22], a technique for measuring similarity of documents in the area of information retrieval. The resulting similarity is represented in a weighted graph where a node represents a method and an edge represents a pairwise relation of methods. The MaxFlow-MinCut algorithm [23] is used to decompose the similarity graph. Their work assumes that there exist ample in-line comments and a naming convention for identifiers instilling the intended context into the name, which is not always the case.

There exists some work on detecting God classes. Chatzigeorgiou *et al.* [2] present a design-based approach for identifying God classes. They use collaboration diagrams to identify objects having significant interactions by observing the number of links between objects. Objects that have high fan-in and fan-out are candidates of God classes. However, that is not always the case, for example facade and proxy classes often have high fan-in and fan-out, but have a single responsibility of delegation. Joshi and Joshi [6] present a lattice-based approach for identifying less cohesive classes. A lattice captures attribute references in methods. They propose seven types of lattices of which five types are cohesive and the other two are less cohesive. A lattice conforming to the less cohesive types is advised to be decomposed. Marinescu [4] proposes metric-based rules for identifying God classes. They observe common symptoms of God classes such as high complexity, low cohesiveness, and frequent access to data in other classes. These symptoms are detected using Weighted Method Count (WMC) [3], Tight Class Cohesion (TCC) [17], and Access To Foreign Data (ATFD) [24]. Daniel *et al.* [1] extend the work by Marinescu using historical data. Classes are classified by frequency of change and the degree of change in size observed in history. The higher frequency and degree of change, the more likelihood of being God classes.

In summary, the existing work on detecting God classes and decomposing responsibilities heavily relies on intra-class information (e.g., attribute-method relationships, internal method dependencies, in-line comments). However, object-oriented systems are collaborative by nature and it is hard to derive precise semantics of methods without considering class interactions. In this work, we use both intra-class information and inter-class information with more emphasis on the latter. Unlike the existing work, the presented approach can be used at both the design level and the code level. At the design level, the approach can be used for class diagrams and sequence diagrams, which enables to detect God classes early in development phase.

3. OVERVIEW OF APPROACH

In this work, we view a class having a purpose for its existence. In the view, we define a responsibility as a set of methods to achieve the intended purpose of the class. Given that, the approach aims at detecting God classes and decomposing their responsibilities to be a single responsibility per class. Figure 1 shows an overview of the approach. In the approach, God classes are detected based on pairwise semantic analysis of methods using Resnik's taxonomy [12]. In the taxonomy, relative similarity for every pair of methods is measured based on the architectural structure of the system using the Semantic Similarity (SS) metric. The resulting similarity captures the structural distance between the paired methods which we use as a base for measuring the semantic similarity of the methods. The closer in distance, the more similar in semantics. The resulting similarity is then refined by considering class relationships which are not taken into account in the structural taxonomy. The refined similarity is then further refined by considering internal method call dependencies within the same class.

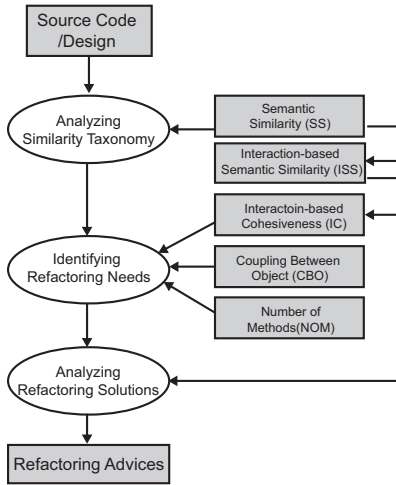


Figure 1: Process Overview

A God class is detected using a set of metrics including Interaction-based Cohesiveness (IC), Coupling Between Object (CBO) [3], and Number Of Methods (NOM) [25]. IC measures the cohesiveness of a class based on similarity of the methods that interact with the class. Method interactions are measured using Interaction-based Semantic Similarity (ISS) based on the structural taxonomy. CBO measures the coupling of a class based on the interactions of the class with other classes, while NOM measures the number of methods defined in a class. Detected God classes are analyzed for their responsibilities using a threshold determined by the average and standard deviation of ISS. The resulting analysis advises a solution for decomposition of responsibilities. We use complete weighted graphs to represent the solution.

4. SEMANTIC SIMILARITY

In this section, we describe analyzing semantic similarity of methods by adopting Resnik's taxonomy [12]. Figure 2 shows an example of the taxonomy capturing the structure of a system in a tree where leafs represent methods and non-leafs represent either classes or (sub)packages. For example, in the figure, methods M1, M2, and M3 are defined in class

C1 and classes C1 and C2 belong to package P2 which is a sub-package of P1. Each node in the tree has its relative distance to other entities. The distance is measured using the following metrics [26]:

$$SS(e_i, e_j) = -\log P(ls(e_i, e_j))$$

where $ls(e_i, e_j)$ is the lowest superordinate of e_i and e_j .

$$P(e) = \frac{|se(e)|}{N}$$

where $se(e)$ is the set of sub-entities of e and N is the total number of nodes.

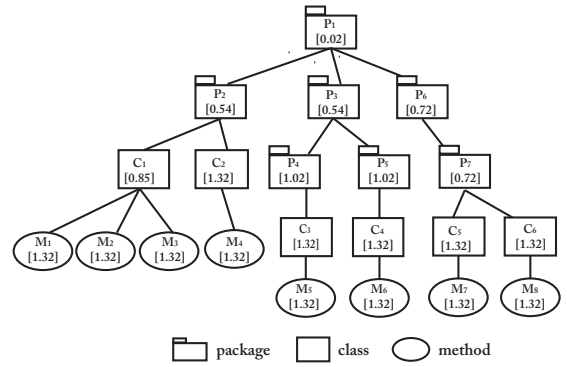


Figure 2: A Semantic Taxonomy of Entities

In Figure 2, the relative distance of M_1 and M_4 is measured 0.54 by $SS(M_1, M_4) = -\log P(ls(M_1, M_4))$ where $ls(M_1, M_4)$ is P_2 . $P(P_2)$ is $\frac{|se(P_2)|}{21}$ where $se(P_2) = \{C_1, C_2, M_1, M_2, M_3, M_4\}$. Thus, $P(P_2) = \frac{6}{21} = 0.29$ and $SS(M_1, M_4) = -\log 0.29 = 0.54$. We use the distance as the semantic similarity of M_1 and M_4 . Similarly, the distance of M_1 and M_5 is measured 0.02. The distances are interpreted that M_1 is more similar to M_4 in semantics than to M_5 since M_1 and M_4 belong to the same sub-package. In the tree, leafs have the maximum similarity since their similarity is measured to themselves. Table 1 shows the results of the taxonomy in matrix. One may consider class libraries in the taxonomy for better results if the results outweigh the overhead.

Table 1: Similarity Matrix

	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇	M ₈
M ₁	1.32	0.85	0.85	0.54	0.02	0.02	0.02	0.02
M ₂		1.32	0.85	0.54	0.02	0.02	0.02	0.02
M ₃			1.32	0.54	0.02	0.02	0.02	0.02
M ₄				1.32	0.02	0.02	0.02	0.02
M ₅					1.32	0.54	0.02	0.02
M ₆						1.32	0.02	0.02
M ₇							1.32	0.72
M ₈								1.32

4.1 Refining Using Class Relationships

The similarities in Table 1 consider only the structural relationships of entities. We refine the similarities by considering types of class relationships which are not captured in the taxonomy. We consider 1) inner relationships, 2) generalizations, 3) aggregations, 4) associations and dependencies, which are in the order of high to low in weight. Inner relationships are weighted 1.5, which is the highest, as the inner class and the outer class have the full access each other. Generalizations are weighted the second 1.4, since child classes can inherit the properties of the parent class, but not all. Aggregations are stronger than associations and dependencies due to the whole-and-part constraint and weighted 1.3. Associations and dependencies are weighted same 1.2 as they can be interchangeably used, although dependencies are a little more limited in use. The range in weights is determined in consideration of the relative influence of class relationships to method similarity. Considering these types with different weights refines the similarities from the structural taxonomy.

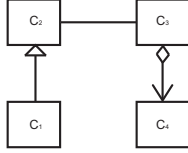


Figure 3: Class Relationships

Table 2 shows the refined similarities for the relationships given in Figure 3. For instance, the similarity of methods M_1 and M_3 in Table 1 is 0.54 and it is refined to 0.76 (0.54×1.4) by considering the generalization in Figure 3. Refined similarities are shown in bold in the table. Note that the refined similarity might be greater than the maximum similarity (1.32) in Table 1, which is conceptually not valid (as nothing can be more similar than itself). To remedy this, the minimum value that makes the maximum similarity greater than the refined value in multiplication is used. This value is referred to as *tapping constant*.

Table 2: Refined Similarities with Weighed Class Relationships

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
M_1	1.32	0.85	0.85	0.76	0.02	0.02	0.02	0.02
M_2		1.32	0.85	0.76	0.02	0.02	0.02	0.02
M_3			1.32	0.76	0.02	0.02	0.02	0.02
M_4				1.32	0.024	0.02	0.02	0.02
M_5					1.32	0.70	0.02	0.02
M_6						1.32	0.02	0.02
M_7							1.32	0.72
M_8								1.32

4.2 Refining Using Method Call Dependencies

We further refine the similarities resulting from Subsection 4.1 by considering method call dependencies within the same class. Having a call dependency between methods belonging to the same class shows a great intimacy and their similarity

is doubled. Method call dependencies are weighed higher than class relationships as method dependencies are more influential to the semantic similarity of methods than class relationships. Suppose method M_1 has a call dependency on method M_2 . Given this, the similarities in Table 2 are refined as shown in Table 3 where the similarity of M_1 and M_2 is refined to 1.7 from 0.85. A tapping constant can be used if the refined value becomes greater than the maximum similarity.

Table 3: Refined Similarities with Weighed Call Dependency within the Same Class

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8
M_1	2.64	1.7	0.85	0.86	0.02	0.02	0.02	0.02
M_2		2.64	0.85	0.86	0.02	0.02	0.02	0.02
M_3			2.64	0.86	0.02	0.02	0.02	0.02
M_4				2.64	0.024	0.02	0.02	0.02
M_5					2.64	0.76	0.02	0.02
M_6						2.64	0.02	0.02
M_7							2.64	0.72
M_8								2.64

5. EXTRACTING RESPONSIBILITIES OF GOD CLASS

The similarities resulting from Section 4 capture only the similarity of methods across classes and do not fully capture the similarities of the methods in the same class. This is because the base taxonomy used in Section 4 is built upon the structure of entities which does not carry any internal information of a class. For this reason, the same similarity is shown in Figure 2 for the methods in the same class (e.g., methods M_1 , M_2 , and M_3 in class C_1 have the same similarity 0.85). The refinement in Subsection 4.2 measures limited similarity of internal methods by considering internal method dependencies. For the full extent of measuring the similarity of the same class methods, we make use of method interactions. That is, the similarity of the same class methods are measured “indirectly” through the similarity of their interacting methods in other classes, which is referred to as Interaction-based Semantic Similarity (ISS). ISS of methods m_i and m_j is measured as follows:

$$\begin{aligned}
 ISS(m_i, m_j) = & mss(F_{in}(m_i), F_{in}(m_j)) \\
 & + mss(F_{out}(m_i), F_{out}(m_j)) \\
 & + SS(m_i, m_j) \\
 mss(ms_i, ms_j) = & \frac{\sum_{m_i \in ms_i} \sum_{m_j \in ms_j} SS(m_i, m_j)}{|ms_i| \times |ms_j|}
 \end{aligned}$$

where $F_{in}(m)$ is the fan-in function of method m returning the set of invoking methods for m and $F_{out}(m)$ is the fan-out function returning the set of invoked methods and $SS(m_i, m_j)$ is the similarity of methods m_i and m_j found in Table 3. For example, suppose that we measure the similarity of M_1 and M_2 in Figure 4. In the figure, the fan-in of M_1 and M_2 is $\{M_4\}$, the fan-out of M_1 is $\{M_6, M_7\}$, and the fan-

out of M_2 is $\{M_7\}$. Based on Table 3, $mss(\{M_4\}, \{M_4\}) = 2.64$, $mss(\{M_6, M_7\}, \{M_7\}) = 1.33$ and $SS(M_1, M_2) = 1.7$, which results in $ISS(M_1, M_2) = 5.67$. The similarity of other pairs can be computed ($ISS(M_1, M_3) = 1.24$ and $ISS(M_2, M_3) = 1.59$).

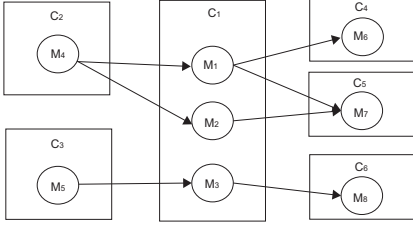


Figure 4: Fan-In and Fan-Out of Methods

5.1 Detecting God Classes

God classes are often large in size and have high coupling with other classes and low cohesion with respect to the similarity of interacting methods [27, 28]. Given this observation, we identify God classes using a set of metrics including Interaction-based Cohesiveness (IC), Number of Methods (NOM) [25], and Coupling Between Objects (CBO) [3]. IC measures the cohesiveness of a class based on the similarity of the methods that interact with the class as follows:

$$IC(c) = \frac{\sum_{m_i \in M} \sum_{m_j \in M} ISS(m_i, m_j)}{|M| \times (|M| - 1)}$$

where M is the set of the methods in class c and $m_i \neq m_j$. For instance, IC of the class C_1 in Figure 4 is measured 2.18. CBO measures the coupling of a class based on the interactions of the class with other classes, while NOM measures the number of methods defined in a class. Using IC, CBO, and NOM, we define the following rule for detecting God classes:

$$GC(S) = \{c \in C \mid (NOM(c) > 3rdQuartile(S)) \wedge (CBO(c) > 3rdQuartile(S)) \wedge (IC(c) < 3rdQuartile(S))\}$$

where C is the set of classes defined in system S . In the rule, IC, NOM, and CBO are set to the 3rd quartile as a default. We use box plots to represent statistical filtering. A detected god class is represented using a complete weighted graph where a node represents a method and the weight on each edge represents the semantic similarity of the paired methods linked by the edge. Figure 5 shows an example graph for the class C_1 in Figure 2.

5.2 Decomposing Responsibilities

God class graphs resulting from Subsection 5.1 are analyzed for decomposition of responsibilities using a threshold. A threshold ϵ is determined as follows:

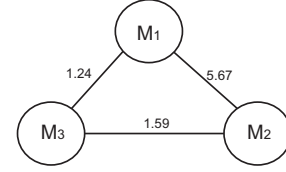


Figure 5: Example of complete weighted graph

$$\epsilon = [\mu - \sigma, \mu + \sigma]$$

if $\mu - \sigma < MinWeight$ then $\epsilon = MinWeight$

if $\mu + \sigma > MaxWeight$ then $\epsilon = MaxWeight$

where μ is the average of weights and σ is the standard deviation of weights. The threshold guarantees the edge of two nodes having the weight (similarity) lower than the threshold to be removed, which splits the God class graph into sub-graphs, each capturing a single responsibility. Figure 6 shows two sub-graphs split from the God class graph Figure 5 by a threshold ranging from 0.37 to 5.30.

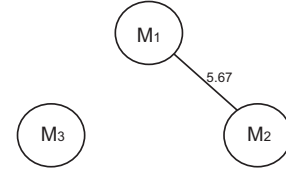


Figure 6: Splitting Responsibilities

6. CASE STUDY: JMETER

We use JMeter v.2.5.1 [29], a Java-based open source for load testing and measuring performance of a server, to evaluate the presented approach. The version used in this study involves 405 packages, 1,623 classes, 9,005 methods, and 30 libraries with 2.5 years of maintenance. The size of the application is 145 KLOC and the average NOM and the average CBO are 8.5 and 11.1, respectively. In applying the approach, the structural taxonomy involves 11,033 entities in total including classes, methods, and packages, which results in a $9,005 \times 9,005$ similarity matrix. Figure 7 shows partial results of the taxonomy and a corresponding matrix is shown in Table 4. In this study, we also consider libraries in similarity analysis. The dashed box in Figure 7 shows a subset of the considered libraries. The great deviation between the minimum value and the maximum value in Table 4 is a hint of the large number of entities used in this study.

The similarities in Table 4 are refined in consideration of class relationships. There are 176 inner class relationships, 187 generalizations, 655 associations/dependencies found. Applying the weights for class relationships in Section 4, the similarities are refined to Table 5.

The similarities in Table 5 are further refined in consideration of method call dependencies which involve 4,066 dependencies. Table 6 shows the refined similarities.

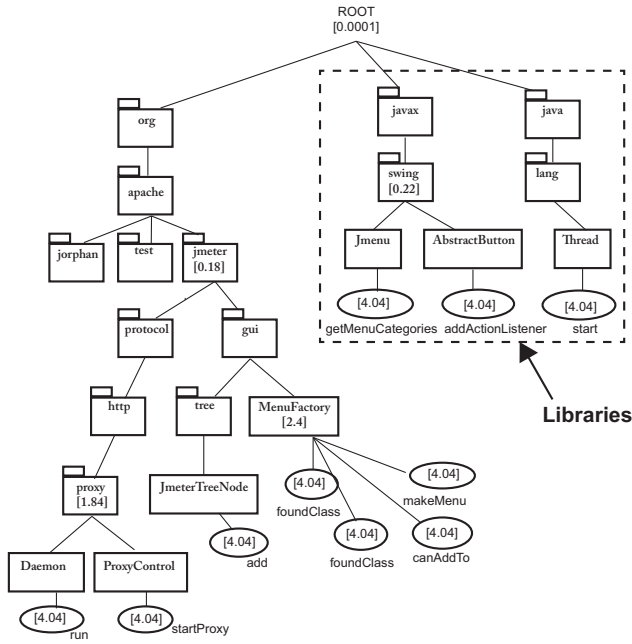


Figure 7: Structural Taxonomy of JMeter v2.5.1

Table 4: Semantic Similarities of JMeter v2.5.1

	run	start Proxy	getMenu Categories	found Class	found Class	canAdd To	make Menu	add	addAction Menu	start
run	4.04	1.84	0.0001	0.18	0.18	0.18	0.18	0.18	0.0001	0.0001
start Proxy		4.04	0.0001	0.18	0.18	0.18	0.18	0.18	0.0001	0.0001
getMenu Categories			4.04	0.0001	0.0001	0.0001	0.0001	0.0001	0.22	0.0001
found Class				4.04	2.7	2.7	2.7	1.34	0.0001	0.0001
found Class					4.04	2.7	2.7	1.34	0.0001	0.0001
canAdd To						4.04	2.7	1.34	0.0001	0.0001
make Menu							4.04	1.34	0.0001	0.0001
add								4.04	0.0001	0.0001
addAction Menu									4.04	0.0001
start										4.04

Based on the resulting similarities in Table 6, the 3rd quartile of CBO, NOM, and IC is measured 9.0, 6.0 and 1.22, respectively, which leads to the following detecting rule:

$$GC(JMeter v2.5.1) = \{c \in C | (NOM(c) > 9.0) \wedge (CBO(c) > 6.0) \wedge (IC(c) < 1.22)\}$$

where C is the set of the classes in JMeter v2.5.1. Figure 8 shows box plots for CBO, NOM, and IC.

By applying the rule, six classes out of 1,623 classes are

Table 5: Refined Similarities of JMeter v2.5.1 with Class Relationships

	run	start Proxy	getMenu Categories	found Class	found Class	canAdd To	make Menu	add	addAction Menu	start
run	4.04	2.2	0.0001	0.18	0.18	0.18	0.18	0.18	0.0001	0.00014
start Proxy		4.04	0.0001	0.18	0.18	0.18	0.18	0.18	0.0001	0.00014
getMenu Categories			4.04	0.00012	0.00012	0.00012	0.00012	0.0001	0.22	0.0001
found Class				4.04	2.7	2.7	2.7	1.6	0.00012	0.0001
found Class					4.04	2.7	2.7	1.6	0.00012	0.0001
canAdd To						4.04	2.7	1.6	0.00012	0.0001
make Menu							4.04	1.6	0.00012	0.0001
add								4.04	0.0001	0.0001
addAction Menu									4.04	0.0001
start										4.04

Table 6: Refined Similarities of JMeter v2.5.1 with Method Call Dependencies

	run	start Proxy	getMenu Categories	found Class	found Class	canAdd To	make Menu	add	addAction Menu	start
run	8.05	2.2	0.0001	0.18	0.18	0.18	0.18	0.18	0.0001	0.00014
start Proxy		8.05	0.0001	0.18	0.18	0.18	0.18	0.18	0.0001	0.00014
getMenu Categories			8.05	0.00012	0.00012	0.00012	0.00012	0.0001	0.22	0.0001
found Class				8.05	5.4	5.4	2.7	1.6	0.00012	0.0001
found Class					8.05	5.4	2.7	1.6	0.00012	0.0001
canAdd To						8.05	2.7	1.6	0.00012	0.0001
make Menu							8.05	1.6	0.00012	0.0001
add								8.05	0.0001	0.0001
addAction Menu									8.05	0.0001
start										8.05

detected as candidate God classes. Table 7 shows the list of the detected classes.

Table 7: Candidate God Classes

God Class	NOM	CBO	ICM
MenuFactory	22	46	1.12
ProxyControl	29	57	0.98
SampleResult	23	252	0.98
AbstractTestElement	24	30	1.20
ProxyControlGUI	22	58	1.08

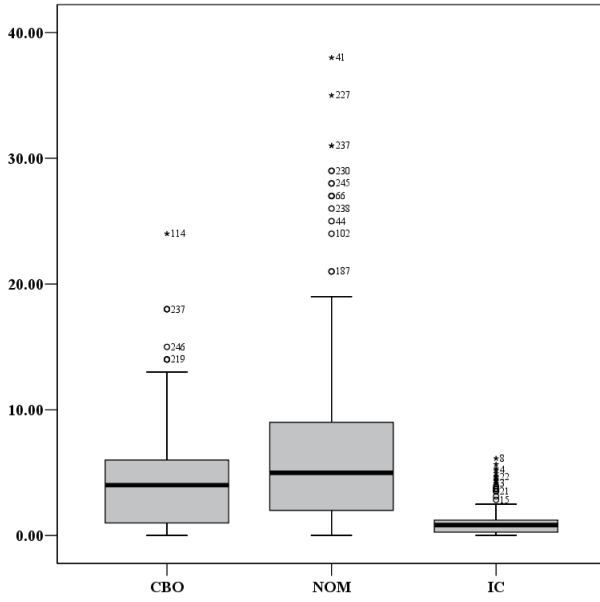


Figure 8: Measures of CBO, NOM, and IC

For each candidate class, a complete weighted graph is built. Figure 9(a) shows the graph for the *MenuFactory* class. The graph involves 22 nodes representing the methods defined in the class and 231 edges connecting the nodes. The threshold for the class ranges from 0.95 to 2.09 and we experimented a threshold ranging from 1 to 2.1 incremented by 0.1. No edge is removed with threshold 1.0 causing no split and 217 edges are removed with threshold 2.1 resulting in three sub-graphs. In a manually verification, threshold 1.5 produces the best result, which decomposes into two sub-graphs with 157 edges removed, each sub-graph representing a single responsibility. One sub-graph involves 18 methods and the other involves 4 methods. Figure 9(b) shows the resulting decomposition. Table 8 shows the threshold ranges used in experiments for each of the candidate God classes.

6.1 Results Analysis

From the case study, we observe three types of God classes shown in Figure 10. Type A has two heterogeneous responsibilities, each having an independent set of fan-in and fan-out interactions, which should be put in a separate class. This is an example of a obvious need for responsibility decomposition. The *MenuFactory*, *SampleResult*, and *AbstractTestElement* classes belong to Type A. The *MenuFactory* class has responsibilities of 1) creating menus and 2) controlling the drag and drop function. While the drag and drop function supports menus, its controlling responsibility is not directly related to menus. The *SampleResult* class involves responsibilities of 1) collecting and storing sample results and 2) measuring the time taken to collect sample results. The collecting and storing functions in the first responsibility is quite heterogeneous to the measuring function in the second responsibility. The *AbstractTestElement* class has responsibilities of 1) configuring properties of tested elements and 2) configuring thread context. The target objects concerned in the two responsibilities are completely different types, and thus the responsibilities share no commonality.

Table 8: Thresholds and ISS Statistics

God Class	Interval of ε	Item	Value
MenuFactory	0.95 ~ 2.09	Mean	1.12
		Std. Dev.	0.97
		Min	0.95
		Max	7.70
ProxyControl	0.86 ~ 2.32	Mean	0.98
		Std. Dev.	1.34
		Min	0.86
		Max	6.87
SampleResult	0.78 ~ 1.50	Mean	0.98
		Std. Dev.	0.53
		Min	0.78
		Max	5.60
AbstractTestElement	0.90 ~ 1.81	Mean	1.20
		Std. Dev.	0.61
		Min	0.90
		Max	4.80
ProxyControlGUI	0.98 ~ 1.48	Mean	1.08
		Std. Dev.	0.40
		Min	0.98
		Max	4.80

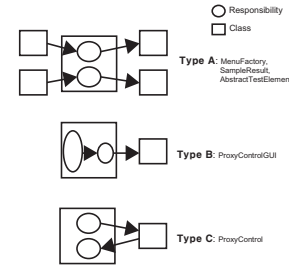


Figure 10: Three Types of Detected God Classes

Type B also involves two responsibilities. However, unlike Type A, the responsibilities in Type B have a dependency. When the responsibilities are split into two classes, the dependency is realized as an association between the classes. The *ProxyControlGUI* class is in Type B. As the name implies, the class involves two dependent responsibilities including 1) creating and controlling GUI and 2) controlling proxies. The proxy responsibility should be separated and put into the *ProxyControl* class which is associated with the *ProxyControlGUI* class. Type B is an example of the *Feature Envy* smell [28] which violates the principle of grouping behaviors by related data and occurs when a method is more interested in being in another class than the current class.

Type C have also two responsibilities that have an indirect dependency via a class. At the class level, the dependency appears as a bidirectional dependency (or association). From an implementation view, a bidirectional dependency is costly as two-way links should be manipulated for creating, accessing, and removing objects, which would not have to be dealt if the responsibilities are separated. The *ProxyControl* class belongs to Type C. The *ProxyControl* class involves responsibilities of 1) starting and stopping

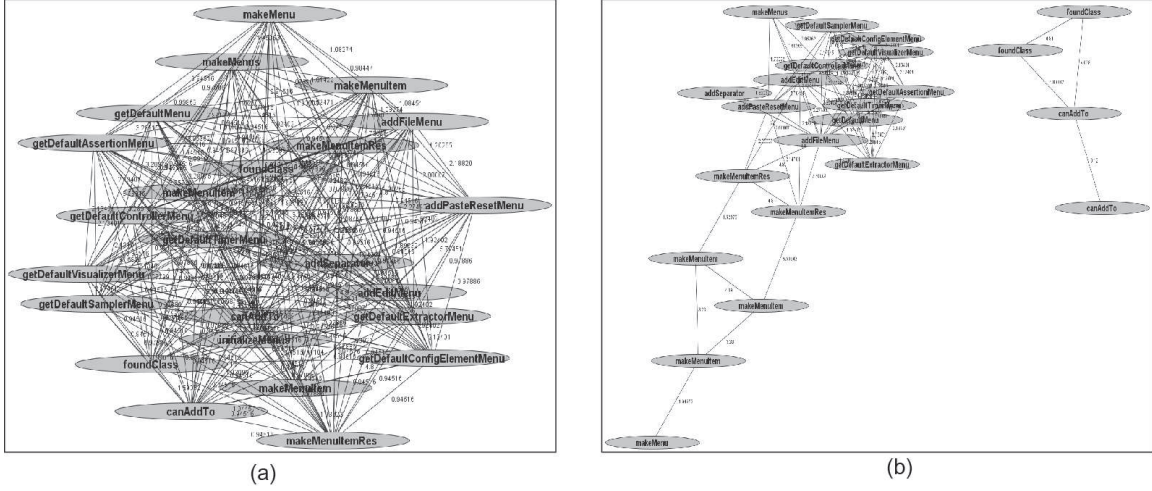


Figure 9: Decomposing Responsibilities of the *MenuFactory* Class

proxies and notifying start and stop of proxies to other objects and 2) receiving the resulting data from proxies and delegating the data to other objects. The first responsibility depends on the *Proxy* class, which in turn depends on the second responsibility. Such a bidirectional association is not a common practice and is often a source of errors [28].

6.2 Verifying Results

We verify the accuracy of the decomposition results by comparing them with manual results produced by two software engineers having 5-year and 2-year industry experience. The engineers manually reviewed the detected classes and their interacting classes for identifying responsibilities. The results of the manual review are shown in Table 9.

Let $R(c) = \{r_1, r_2, \dots, r_n\}$ be the set of manually identified responsibilities of class c where $r_i = \{m_1, m_2, \dots, m_p\}$ is a set of methods. Let $R'(c) = \{r'_1, r'_2, \dots, r'_m\}$ be the set of the responsibilities identified by the presented approach where $r'_i = \{m'_1, m'_2, \dots, m'_q\}$ is a set of methods. For $r_i \in R$ and $r'_j \in R'$, the best matching responsibility in R' is $r'_{best} = \operatorname{argmax} \frac{|r'_j \cap r_i|}{|r'_j \cup r_i|}$. Given that, the accuracy of the presented approach can be measured using the following precision, recall, and F-Measure:

- Precision: The number of correctly identified methods of a responsibility r_i to its best matching responsibility r'_{best} over the number of the identified methods of r_i .

$$\text{Precision} : P(r_i) = \frac{|r_i \cap r'_{best}|}{|r'_{best}|}$$

- Recall: The number of correctly identified methods of a responsibility r_i to its best matching responsibility r'_{best} over the number of the defining methods of r_i .

$$\text{Recall} : R(r_i) = \frac{|r_i \cap r'_{best}|}{|r_i|}$$

- F-Measure: A composite measure of $P(r_i)$ and $R(r_i)$ for responsibility r_i .

$$F - \text{Measure} : F(r_i) = \frac{2 \cdot P(r_i) \cdot R(r_i)}{P(r_i) + R(r_i)}$$

$$F(GC) = \frac{2 \cdot \frac{\sum_{r_i \in R} P(r_i)}{|R|} \cdot \frac{\sum_{r_i \in R} R(r_i)}{|R|}}{\frac{\sum_{r_i \in R} P(r_i)}{|R|} + \frac{\sum_{r_i \in R} R(r_i)}{|R|}}$$

Figure 11 shows the results of F-Measure for the detected God classes per change of threshold. The results in the graph are capricious, which indicates a high deviation of ISS. In fact, the likelihood of being split for a God class graph increases as the deviation of edge weights increases. Therefore, the results indicate a high likelihood of decomposition for the detected classes. The graph also show high accuracy of the results in the threshold range of 1.3 and 1.5.

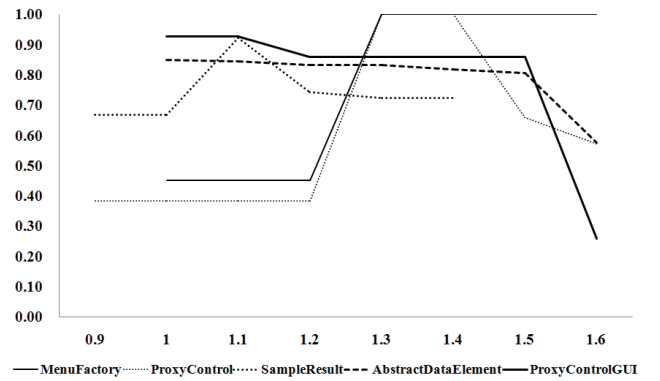


Figure 11: F-Measure for Detected God Classes

In Table 9, the average of the best F-Measures for the detected God classes is 0.919 which is promising. In particular, the responsibilities of the *MenuFactory* and *ProxyControl* class are identified exactly the same as the manually found ones. In this study, the accuracy of the detecting rule is not

Table 9: The best accuracy measure

God Class	Responsibility	#Method			Precision	Recall	F-Measure
		Engineers extracted	Automatically extracted				
			Found	Matched			
MenuFactory (1.3)	(1) Creating/Controlling Menu	18	18	18	1	1	1
	(2) Controlling Drag/Drop Functionality	4	4	4	1	1	1
ProxyControl (1.3)	(1) Controlling Proxies	11	11	11	1	1	1
	(2) Receiving/Delivering Result from Proxies	18	18	18	1	1	1
SampleResult (1.0)	(1) Creating and Having Result	16	16	16	1	1	1
	(2) Measuring Time	9	10	8	0.8	0.889	0.842
AbstractTest Element (1.1)	(1) Configuring Properties of Test Elements	26	25	25	1	0.961	0.98
	(2) Configuring Thread Context	2	1	1	1	0.5	0.667
Proxy Control GUI(1.1)	(1) Creating/Controlling GUI	11	11	11	1	1	1
	(2) Controlling Proxies	14	9	8	0.889	0.571	0.696
				average	0.969	0.892	0.919
				std. dev.	0.069	0.192	0.134

measured as it is not feasible to identify the actually existing God classes in JMeter due to subjectivity and the large number of classes (1,623).

7. CONCLUSION

In this paper, we have presented a semantic-based approach for detecting God classes and decomposing their responsibilities. The approach measures semantic similarity of methods using inter and intra-interactions of methods and class relationships. The resulting similarity is used as a basis for identifying God classes using the NOM, CBO, and IC metrics. Detected God classes are represented in a weighted graph for responsibility analysis and decomposition. Responsibilities are identified based on relative semantic similarity of defined methods in the God class to the similarity of their interacting methods in other classes. Responsibilities are decomposed by a threshold determined by the average and standard deviation of ISS. We evaluate the approach using JMeter v2.5.1.

The presented approach does not require code details (e.g., attribute-method references), and therefore can be used at both design level and code level. In this paper, we did not consider constructors, getters, and setters since they are not captured at design level. At code level, however, constructors may be a subject of interest in detecting God classes and decomposing responsibilities. Getters and setters at code level can help to improve the precision of similarity if they are referenced in other methods, which basically captures attribute-method references. Similarity precision can be further improved if libraries are considered in the taxonomy. However, it involves an overhead and is recommended only when the accuracy of the expected results outweigh the overhead.

Acknowledgements.

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised

by the NIPA. (National IT Industry Promotion Agency(NIPA-2011-(C1090-1131-0008)))

8. REFERENCES

- [1] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu. Using history information to improve design flaws detection. In *Proceedings of European Conference on Software Maintenance and Reengineering*, pages 223 – 232, 2004.
- [2] A. Chatzigeorgiou, S. Xanthos, and G. Stephanides. Evaluating object-oriented designs with link analysis. In *Proceedings of International Conference on Software Engineering*, pages 656–665, 2004.
- [3] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [4] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 350 – 359, 2004.
- [5] F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of European Conference on Software Maintenance and Reengineering*, pages 30 –38, 2001.
- [6] P. Joshi and R.K. Joshi. Concept analysis for class cohesion. In *Proceedings of European Conference on Software Maintenance and Reengineering*, pages 237 –240, 2009.
- [7] K. Cassell, P. Andreae, L. Groves, and J. Noble. Towards automating class-splitting using betweenness clustering. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, pages 595 –599, 2009.
- [8] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander. Decomposing object-oriented class modules using an agglomerative clustering technique. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 93 –101, 2009.

- [9] A. De Lucia, R. Oliveto, and L. Vorraro. Using structural and semantic metrics to improve class cohesion. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 27–36, 2008.
- [10] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *Proceedings of IEEE/ACM international conference on Automated software engineering*, pages 151–154, 2010.
- [11] G. Bavota, A. De Lucia, and R. Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011.
- [12] P. Resnik. Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 448–453, 1995.
- [13] S. Chidamber and C. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings of International Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 197–211, 1991.
- [14] W. Li and Henry S. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122, 1993.
- [15] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, 1995.
- [16] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.
- [17] J. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of Symposium on Software reusability*, pages 259–262, 1995.
- [18] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 133–142, 2005.
- [19] L. Briand, S. Morasca, and V. Basili. Defining and validating high-level design metrics. *Technical Report No. UMIACS-TR-94-75*, 1994.
- [20] P. Sneath and R. Sokal. *Numerical taxonomy: the principles and practice of numerical classification*. Series of books in biology. W. H. Freeman, 1973.
- [21] M. Girvan and M. Newman. Community structure in social and biological networks. *the National Academy of Sciences*, 99(12):7821–7826, 2002.
- [22] S. Deerwester, S. Dumais, T. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [23] T. Cormen, C. Leiserson, R. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, the 2nd revised edition edition, 2001.
- [24] R. Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, pages 173–182, 2001.
- [25] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.
- [26] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 2001.
- [27] W. Brown, R. Malveau, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.
- [28] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.
- [29] JMeter. <http://jmeter.apache.org/>.